Throughput constrained parallelism reduction in cyclo-static dataflow applications

Sergiu Carpov, Loïc Cudennec, Renaud Sirdey

International Conference on Computational Science, ICCS 2013

Throughput constrained parallelism reduction in cyclo-static dataflow applications

Sergiu Carpov*, Loïc Cudennec, Renaud Sirdey

CEA, LIST,
Embedded Real Time Systems Laboratory,
Point Courrier 94, 91191 Gif-sur-Yvette Cedex, France.

Abstract

This paper deals with semantics-preserving parallelism reduction methods for cyclo-static dataflow applications. Parallelism reduction is the process of equivalent actors fusioning. The principal objectives of parallelism reduction are to decrease the memory footprint of an application and to increase its execution performance. We focus on parallelism reduction methodologies constrained by application throughput. A generic parallelism reduction methodology is introduced. Experimental results are provided for asserting the performance of the proposed method.

Keywords: Parallelism reduction; dataflow programming; CSDF; actor fusion

1. Introduction

Nowadays, much effort is dedicated to the study of many-core computing systems, beginning with hardware architecture design issues and ending with software programmability aspects. The main difficulty of efficient utilization of parallel systems resides in their programming, both in terms of conception time and as well as execution performance. The dataflow model of computation has been purposely introduced to facilitate parallel systems programming.

A dataflow application is a network of *actors* communicating through unbounded, unidirectional FIFO channels and exclusively through this channels. There are many instantiations of dataflow programming models (SDF, CSDF, BDF, etc.) [1]. One of these is the cyclo-static dataflow (CSDF) graph [2]. CSDF model is particularly well suited for programming embedded systems because several important application properties (absence of deadlock, bounded memory execution etc.) can be proven. Let A be a CSDF application. The main goal of this study is to obtain a new application A' which is semantically equivalent to the initial application (i.e. application A' has to produce the same results when applied to the same input data) but with fewer "parallelism" in it. We call this action *parallelism reduction*. We are allowed to change the CSDF network topology (add/delete actors and communication channels) as long as the modified application preserves its semantics. The actors are considered black boxes so that we are not allowed to modify actor code and interface.

The advantages of parallelism reduction are: memory footprint of application binaries decreases (less redundancy in code/data loading), program compilation is faster, scheduling overhead is lower and by consequence system times

*Corresponding author
Email address: sergiu.carpov@cea.fr (Sergiu Carpov)

are smaller etc. These advantages are even more important in embedded systems where on-chip memory size is small and scheduling algorithms are sensible to the number of actors. Unlike applications dedicated to high performance computing, the sizing of applications in embedded systems does not only rely on performance goals. It has to meet the following requirements: (i) being parallel enough in order to offer the desired application throughput and (ii) being small enough in order to fit the memory footprint of the target chip. Therefore, application sizing in embedded systems can be seen as a trade-off between performance and memory consumption. As for every complex application design, we think this trade-off should be transparently managed by the compiler.

ΣC is a programming language which allows to easily implement CSDF applications, refer to [3] for more details. Besides providing tools and methods for dataflow graph description, the ΣC language introduces a series of system actors which intend to facilitate the programmers' job. The system actors allow to read, write and reorganize streams of data tokens. One can distinguish the *split* and the *join* system actors. The main purpose of the ΣC language is to make abstraction of the used hardware architecture. That is to say the developer expresses the maximum level of parallelism in the conceived application and the compiler adapts (reduces) it to the specifications of the chosen architecture. This is quite a different and uncommon approach regarding regular parallel programming languages. Our work is particularly aimed at parallelism reduction in ΣC applications.

The parallelism reduction problem is not well known to the literature. One can mention the paper [4] to which our work resembles the most. The authors describe a pattern substitution based method for parallelism reduction in ΣC applications. Initially, the instantiations of a predefined set of patterns are matched in the application. Afterwards each instantiation is replaced by an equivalent pattern of smaller size. The size of the replacement pattern is derived from a global reduction factor. Their goal is to bound the number of actors per processing core to a predefined limit. While reducing the memory footprint, this approach does not ensure that the execution throughput is preserved.

Although in a different context, the authors of the StreamIt [5] dataflow language mentioned about the significance of parallelism reduction in dataflow applications. They use task fusion [6] to adapt application granularity to the target hardware architecture. A limitation of their work is that the tasks must be either horizontal neighbors (pipeline) or vertical neighbors (split-join) in order to be fusioned. Task fusion is not limited to equivalent tasks.

Similar approaches have been studied in the field of FPGA synthesis. The authors of [7, 8] propose a pattern-matching based method for reducing FPGA resource usage at the price of an increased circuit latency. In the work [9] several heuristics for maximizing FPGA resource sharing are studied.

In this paper we introduce a generic parallelism reduction method. The proposed method does not depend on a predefined set of patterns and is not limited to horizontal or vertical actor fusion. It reduces the inherent application parallelism in function of actor execution times and application throughput constraints. In what follows we firstly introduce some preliminary notions and the context of our problem, afterwards we describe the generic parallelism reduction methodology and provide some computational results, finally, the last section concludes the paper.

2. Preliminaries

A CSDF application is denoted by $A = (T, E)$ where T is the set of actors and E is the set of communication channels. The smallest unit of data which traverses a channel is called a *token*. The actors have several input and output ports. The number and the type of these define the actor *interface*. An actor is simply a piece of code that reads data from input ports, treats it and produces data on output ports. A communication channel connects two actor ports. Each actor is executed repeatedly in a finite number of cycles. A cycle can begin only when the required quantity of tokens is present on its input channels. In each cycle a different quantity of tokens is consumed/produced on each input/output channel by the actor. The quantity of tokens consumed/produced on a channel is, respectively, referred to as the *cycle input degree* and as the *cycle output degree*. The *input/output degree* of an actor on a channel is the aggregated quantity of cycle input/output degrees on this channel.

A vector $\vec{r}$, $\vec{r} = [r_1, r_2, \ldots, r_{|T|}]$, is a *repetition vector* for a CSDF application if r_t gives the number of invocations an actor t must perform until the application returns to its initial state. By application state we mean equal token number on each channel. The repetition vector can be found by solving CSDF balance equations [2]. This vector plays an important role in CSDF graph consistency, liveness (absence of dead-locks) and static scheduling.

Often in the context of embedded systems a dataflow application must be able to treat input data streams with a given *throughput* (e.g. audio/video streams with predefined bit-rates). System throughput is defined as the quantity

of data treated per unit of time. Hereafter we use an equivalent measure - actor *goal frequency* - which represents the number of actor executions per time unit. Let ξ_t denote the goal frequency of an actor t. Without loss of generality we suppose that only a single actor p has a predefined goal frequency ξ_p (usually input/output actor). It is obvious this goal frequency propagates to other application actors. The goal frequencies of actors are proportional to repetition vector values. Suppose, for example, an application in which an actor has a repetition value 2 and have to be executed 30 times per second, then an actor with a repetition value 3 will need to be executed 45 ($= {}^3/_2 \cdot 30$) times per second. The *application goal frequency* ξ_A is defined as the ratio between the predefined goal frequency ξ_p and the repetition value r_p of the corresponding actor, i.e. $\xi_A = {}^{\xi_p}/_{r_p}$. In this case the goal frequency of any actor t can be computed using relation $\xi_t = \xi_A \cdot r_t$.

Goal frequencies are important for the execution of embedded dataflow applications. They permit to assert a priori if an application can be executed or not on a given hardware platform. This is done by comparing the goal frequency of each actor to its execution time inverse. An application $A = (T, E)$ can be executed only if relation (1) is verified for any actor t, $t \in T$, where τ_t denotes actor t execution time. This constraint results from the fact that the goal frequency of an actor must be smaller than its maximal execution rate (execution time inverse).

$$\xi_t \leq \tau_t^{-1} \tag{1}$$

In ΣC applications instances of the same actor are called *equivalent actors*. Equivalent actors perform the same computation but on different data streams. Two or more equivalent actors can be merged together[1]. The corresponding input and output data streams are "merged". The goal frequency of the resulting actor is the sum of initial goal frequencies. This goal frequency must satisfy relation (1). The merge of equivalent actors represents a parallelism reduction method. The semantics of the modified application does not change. Parallelism reduction is particularly well suited for ΣC applications because of the high parallelism level that the programmer is able to express.

3. Generic parallelism reduction

In this section we describe a generic parallelism reduction methodology based on equivalent actor merge. The inherent data parallelism present in a CSDF application is reduced to a level at which goal execution constraints remain satisfied.

3.1. Split and join actors

A *split* is an actor with one input port and n output ports. To each output port k is associated a production rate p_k, $p_k \in \mathbb{N}^+$. A split is executed in n cycles. During the k-th execution cycle the split takes p_k tokens from its input port and sends them to the k-th output port. After the n-th p_n-token packet has been transfered the process starts over again (round-robin behavior). A split with n outputs and different production rates is denoted $S\,(n, p_1, p_2, \ldots, p_n)$.

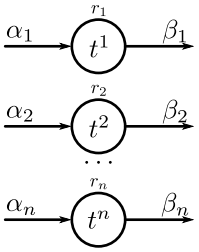
A *join* is an actor which has n input ports and a single output port. As previously, for each input port k of the join a consumption rate c_k, $c_k \in \mathbb{N}^+$, is defined. A join actor is executed in n cycles. At cycle k the join takes c_k tokens from the k-th input port and sends them to its output port. After the n-th c_n-token packet has been transfered the process starts over again (round-robin behavior). A join with n inputs and different consumptions rates is denoted $J\,(n, c_1, c_2, \ldots, c_n)$.

3.2. Merge of equivalent actors

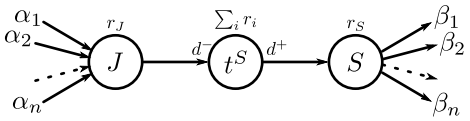
Let S, $S = \left\{t^1, \ldots, t^n\right\}$, be a set of equivalent actors. Suppose r_i is the number of times actor t^i is executed during one iteration of the CSDF application, i.e. r_i is the repetition vector value for actor t^i. Without loss of generality we suppose that the actors have a single input port and a single output port. The input and output degrees[2] of the ports are d^- and respectively d^+. Also let us denote the channels connected to actors input ports with $\alpha_1, \ldots, \alpha_n$ and respectively to actors output ports with $\beta_1, \ldots, \beta_n$. Refer to Figure 1a for an illustration.

[1] We must note that equivalent actors cannot be merged together unless they are stateless. An actor is stateless if it uses data only from the input ports and does not have state variables.

[2] The input and output degrees of all actors coincide because they are equivalent.



(a) A set of equivalent actors.



(b) Merged actors.

Figure 1: Merge of a set S of equivalent actors.

The set of equivalent actors can be merged into a single actor denoted t^S. This is done by (i) removing the actors $t^1, \ldots, t^n$ and adding an actor t^S which has the same code and interface, (ii) the input channels $\alpha_1, \ldots, \alpha_n$ are *time-multiplexed* by a join actor $J(n, c_1, \ldots, c_n)$ and (iii) the output channels $\beta_1, \ldots, \beta_n$ are *time-demultiplexed* by a split actor $S(n, p_1, \ldots, p_n)$. In Figure 1b a merge of equivalent actors is illustrated. In what follows we describe how the parameters, c_i, p_i, of join and split actors are found.

After the merge operation, the number of data tokens taken from an input channel α_i by the join actor must induce an integral number of actor t^S executions. Otherwise an incoherence in the contents of data packets treated before and after the merge will be created. In order to avoid this incoherence d^- has to be a divisor for c_i. Let k_i denote the quotient of this division, i.e. $c_i = k_i \cdot d^-$. The data produced on channel β_i must originate from executions of actor t^S on data consumed from the channel α_i, so to k_i consumptions on channel α_i must correspond k_i productions on channel β_i. The split parameters must verify $p_i = k_i \cdot d^+$.

The rates at which data is consumed from channels α_i and produced to channels β_i should not change after the merge operation, otherwise an incoherency in the application is created. Consumption rate before the merge operation on channel α_i is $r_i \cdot d^-$. After the merge, the consumption rate is $r_J \cdot c_i = r_J \cdot k_i \cdot d^-$, here r_J denotes the repetition value of the join actor. These two rates (before and after merge) must be equal (i.e. $r_i \cdot d^- = r_J \cdot k_i \cdot d^-$). The repetition value r_J must satisfy relations:

$$r_i = k_i \cdot r_J, \ \forall i = 1, \ldots, n,$$

thus r_J have to divide all r_i-s.

The quotients k_i are computed given a value for r_J. The best choice for r_J is the greatest common divisor of repetition values:

$$r_J = \gcd_i (r_i)$$

We could also use the trivial solution $r_J = 1$. The disadvantage of the latter, compared to the greatest common divisor one, is that join input degrees c_i will have larger values and consequently the CSDF edge buffers will potentially augment in size.

To sum up, initially we compute the quotients k_i using equation (2) and then the input degrees $c_i = k_i \cdot d^-$ of join actor and the output degrees $p_i = k_i \cdot d^+$ of split actor for any $i = 1, \ldots, n$.

$$k_i = \frac{r_i}{\gcd_j \left(r_j\right)} \tag{2}$$

For the case of equivalent actors with more than one input and/or output ports the procedure is practically the same, except that a join actor is added for each input port and a split actor for each output port. The input and output degrees of join and split are computed equivalently using k_i.

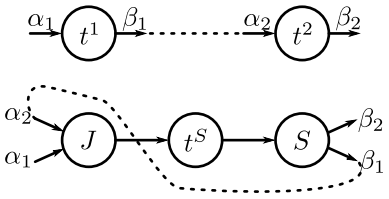


Figure 2: Deadlock due to invalid multiplexing order.

3.3. Parallelism reduction methodology

As we have seen in the previous subsection a set of equivalent actors can be merged together without changing application semantics. Here, we describe a methodology for reducing the parallelism of an entire CSDF application using equivalent actor merge operations.

Initially application actors are partitioned into sets of equivalent actors, i.e. instances of the same actor are grouped together. The sets of equivalent actors containing only one actor are directly discarded as no parallelism reduction is possible. The sets containing more than one element are kept for a potential parallelism reduction.

Let $S = \left\{t^1, \ldots, t^n\right\}$ be a set of equivalent actors found above. The set S is split into m partitions $S_1, S_2, \ldots, S_m$, which satisfy goal execution frequency constraints. The partitioning procedure is described in more details in the next subsection.

The actors from each partition S_i are merged into a single actor t^{S_i}. The corresponding input and output ports are time-multiplexed and respectively time-demultiplexed. The multiplexing order is $i_1, i_2, \ldots, i_{|S_i|}$. This order should verify the following property: for any couple of actors t^{i_k} and t^{i_l} if $t^{i_k} \in \text{anc}\left(t^{i_l}\right)$ then $i_k < i_l$. Here $\text{anc}(t)$ is the set of ancestors of actor t in the CSDF graph[3]. If the last condition is not satisfied the application will deadlock. For example, suppose the application illustrated in Figure 2 (top). The reduction illustrated in the bottom part uses an invalid multiplexing order. The join actor deadlocks because no data is available on channel α_2.

One way to interpret the parallelism reduction described above is that in the resulting CSDF application the actors belonging to a set S_i are serialized. The sequential execution order of these actors is given by the indexes $i_1, i_2, \ldots, i_{|S_i|}$. One can choose this sequence such that an objective function is optimized, refer to [10] for a possible model which aims data reuse optimization.

The result of the parallelism reduction is the replacement of actors $t^1, \ldots, t^n$ with actors $t^{S_1}, \ldots, t^{S_m}$, thus a reduction of parallelism from n to m. At the same time the goal frequency constraints remain satisfied in the modified application.

3.4. How to choose the partitions

Earlier we have supposed that the sets of equivalent actors are already divided into partitions, but we did not provide any details on how to choose these partitions. In this section a bin-packing based approach for partitioning a set of equivalent actors is described.

Suppose a set S of n equivalent actors $t^1, t^2, \ldots, t^n$ is given. Each actor t^i has a repetition value r_i defined. The goal frequency ξ_i of this actor is computed using relation $\xi_i = \xi_A \cdot r_i$ (recall that ξ_A is the application goal frequency). The execution times of the actors are the same and denoted by τ. The problem we search to solve consists in finding a minimal integer m and a m-partition $S_1 \cup S_2 \cup \ldots \cup S_m$ of S such that relation (3) is verified.

$$\sum_{t \in S_k} r_t \leq \frac{1}{\xi_A \cdot \tau}, \ \forall k = 1, \ldots, m \tag{3}$$

The repetition value of a composed actor equals to the sum of repetition values of its elements, $r\left(t^S\right) = \sum_{t \in S_k} r_t$, hence the goal frequency of the composed actor is $\xi_A \cdot r\left(t^S\right) = \xi_A \cdot \sum_{t \in S_k} r_t$. Inequality (3) is inferred from the last equality and

[3] If the CSDF graph is not acyclic then a similar method, based on CSDF graph unfolding, can be used to define the multiplexing order.

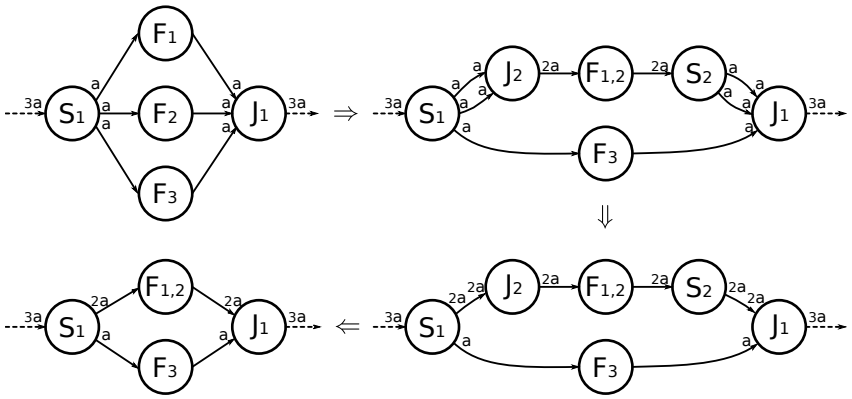


Figure 3: Elimination of parallelism reduction redundancies. Above some channels are indicated production and consumption rates of corresponding actor ports.

from relation (1). This constraint assures that the composed actors can be executed without violating the throughput constraint.

The problem defined in this way is a *bin packing* problem. It is well known and has been extensively studied in the past decades. For a complete survey on bin packing models refer to [11]. In general the bin packing problem is $\mathcal{NP}$-hard [12] but can be solved in polynomial time when the number of item sizes is bounded[4]. In our case this corresponds to a bounded number of actor repetition values.

For real life dataflow applications it is unrealistic to have a large number of different repetition values for equivalent actors. In extremis we can restrict the notion of equivalent actors by considering actor repetition value as a discriminating parameter when the sets of equivalents actors are found (see the initial phase in Subsection 3.3). So it is conceivable to find optimal solutions using this method for real world applications.

A disadvantage of the bin packing model is that it is not straightforward to include other measures in the objective function than the number of used bins. It would be interesting to partition the actors more intelligently (maximize the data reuse for example) and not only with the number of partitions minimization objective.

3.5. Parallelism reduction induced redundancies

The use of the parallelism reduction methodology described above has pointed out that redundancies are introduced in some ΣC dataflow applications. In the next paragraph we show that these redundancies can be removed without any implication on application semantics.

In several applications, after the parallelism reduction, redundant edges are created between split and join actors. For example, lets examine a simple application illustrated in the top-left corner of Figure 3. Suppose that two equivalent actors **F1**, **F2** are merged into a single one **F1,2**. The input channels of these actors are time-multiplexed using join **J2** and respectively the output channels are time-demultiplexed using split **S2**. The obtained application is shown in the top-right corner of the figure.

Consecutive links between a split and a join actor with equal consumption and production rates can be merged into a single link. The new production (consumption) rate is equal to the sum of initial production (consumption) rates. In our example the links between actors **S1**, **J2** and respectively between actors **S2**, **J1** are merged and the resulting application is illustrated in the bottom-right corner. The last application can be further optimized by shortcutting splits and joins with only one input and one output (these actors being useless) with a link. See the bottom-left corner of Figure 3.

From now on we suppose that the application redundancies are removed if present.

[4]Even when this number is not bounded several algorithms provide good worst-case performance ratios.

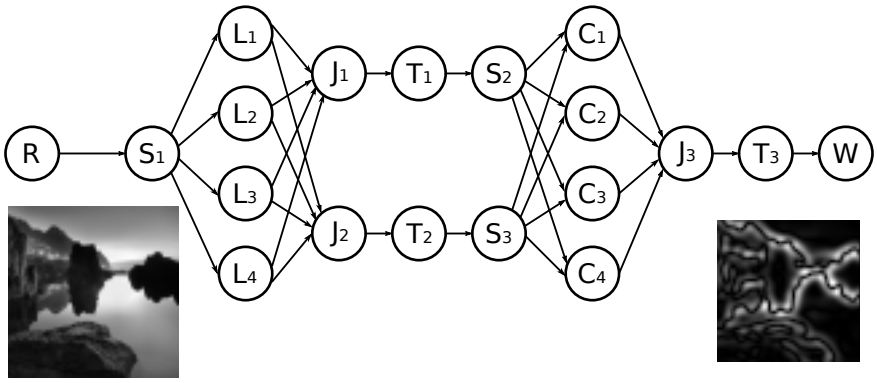


Figure 4: LoG edge detection application.

4. Computational results

In this section we examine the impact of parallelism reduction on the compilation chain for ΣC applications and on the execution of ΣC application binaries. The tests have been performed on a mid-range laptop with an Intel P8600 processor (2 cores).

As sample application we use the Laplacian of Gaussian (LoG) edge detection algorithm [13] with an image smoothing step. The smoothing step is done by a convolution of the input image with a Gaussian kernel. Image edges are found by convolving the smoothed image with a LoG kernel. The order in which the convolutions are applied does not matter because the convolution operator is commutative. So, we can smooth the image after the edges are found. Furthermore, if the convolution kernel is separable into horizontal and vertical components we can apply the convolution by lines and after by columns, which is our case.

This edge detection algorithm has been implemented in ΣC language. The application graph is illustrated in Figure 4 together with a sample input and output image. We use 11×11 convolution kernels and input image sizes are 64×64. The *available parallelism* is the number of line, **Li**, or column, **Ci**, filters (horizontal and vertical convolutions) which can be executed in parallel. For example, the application illustrated in Figure 4 has an available parallelism of 4. We have generated four versions of edge detection application with an available parallelism of 2, 4, 8 and respectively 16. Goal frequency of line (column) filters is fixed in such a way that their number cannot drop below 2 after the parallelism reduction.

Two types of parallelism reductions (obtained by manually modifying the equivalence of transpose actors) are used for exemplification purposes. In the first one, denoted partial reduction, only the first two transpose actors, **T1** and **T2**, are merged together. In the second reduction, denoted full reduction, all three transpose actors are merged. The partial reduction is illustrated in Figure 5 and the full reduction in Figure 6. In these applications the transpose actors are merged and their input and output channels are multiplexed using join actor **J4** and split actor **S4**. For the full reduction case it can be observed that the order in which the transpose input channels are multiplexed prevent a deadlock creation. Split and join actors used to time-(de)multiplex input and output channels of filter actors were deleted because they were parallelism reduction induced redundancies. The goal frequency constraint remains respected as the number of line and column filters is 2 after the reduction. Although the final application graphs have the same structure they differ by the granularity of split and join actors, **S1**, **S2**, **S3**, **J1**, **J2**, **J3**, which are respectively 64, 128, 256, 512 for available parallelism of 2, 4, 8 and 16. Here the granularity refers to cycle input/output degrees of split and join actors.

4.1. Compilation

In the first experiment we examine the influence of parallelism reduction methodology on compilation times of ΣC applications. We have generated 12 versions of LoG edge detection application. They differ in function of the available parallelism (2, 4, 8 and 16) and parallelism reduction type (without, partial and full). Each application version has been compiled and the compilation times have been saved. We have repeated this procedure for 50 times

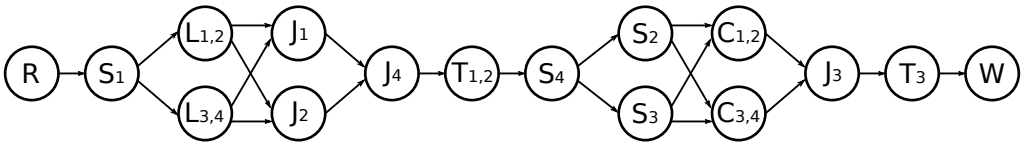


Figure 5: Partial reduction of LoG edge detection application.

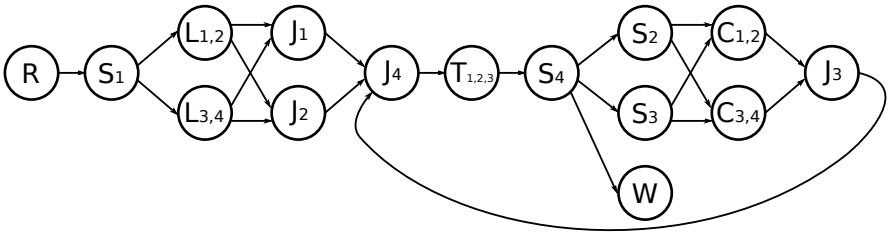


Figure 6: Full reduction of LoG edge detection application.

in order to obtain statistically representative results. Average compilation times are computed for each version. The results are reported in Figure 7.

Above each bar in the chart the number of actors in the final application is shown. These numbers are reported to have an idea about the ratio between compilation time and actor number. This ratio is almost constant and does not depend on application version.

In the case when the available parallelism is equal to 2 only the transpose actors are merged. We can observe that despite the fact that the number of actors does not decrease the compilation times are smaller for partial and full reduction versions. This is due to a lighter complexity of system actors compilation. Compilation times for the partial and full reduction versions do not differ and are almost equal for any available parallelism. In the extremal case (available parallelism of 16) the parallelism reduction procedure allows to compile the same application at least two times faster. We can conclude that even if the application developer expresses an exhaustive level of parallelism the compilation process will take the same time as for an application with an "optimal" level of parallelism.

Besides compilation times we have also examined the size of the obtained binaries for each application version. The ratios of binary sizes after parallelism reduction to the sizes of the initial application are given in Table 1. For partial and full reduction the binaries have approximatively the same size. The parallelism reduction decreases the binary size in all of the studied application versions. In the extreme case (available parallelism is 16) the binary of a non reduced version is more than 3 times larger than the binary of a reduced one.

4.2. Execution

We have employed the posix-thread back-end of the ΣC compiler for generating binaries of the edge detection application. This back-end is used for simulating applications functional behavior. The last fact makes impossible the use of frequency goal constraint for guiding parallelism reduction. Nevertheless we have used the 16 versions of LoG edge detection application generated in the previous subsection for comparing execution times on a general purpose computer. We think that the use of a real embedded platform is relevant but not mandatory to show how the parallelism reduction optimizes the application (binary size is reduced while the execution throughput constraint remains satisfied).

Available parallelism	2	4	8	16
No reduction/partial reduction	1,07	1,37	1,97	3,20
No reduction/full reduction	1,15	1,47	2,11	3,44

Table 1: Binary size ratios before and after parallelism reduction for the edge detection application.

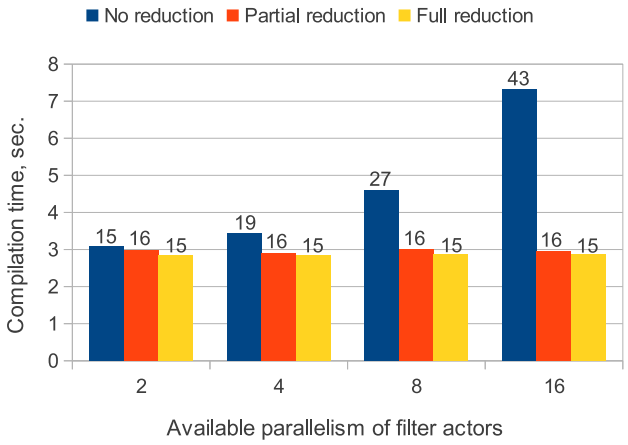


Figure 7: Compilation times of edge detection application.

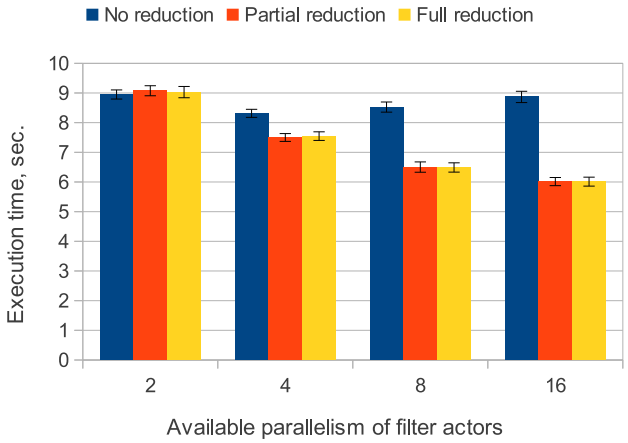


Figure 8: Execution times of edge detection application.

ΣC dataflow applications assume endless execution. In our tests the application is interrupted after the edge detection algorithm is executed one thousand times. As previously each application version is executed for 50 times and execution times are saved. In Figure 8 are illustrated the average execution times we obtain together with corresponding standard deviations.

Application versions without reduction have comparable execution times despite that the application graphs have different sizes. This fact is true because the same amount of information is treated by all the applications. The small differences in execution times are due to the posix execution system (actor scheduling, etc.). As we can see the best compromise is achieved for an available parallelism of 4.

As for the optimized versions of the application (partial and full reduction), with the increase of available parallelism the execution performance is better. In the application with an available parallelism of 16 both parallelism reductions lower the execution time by approximatively one-third (30%). The decrease of execution times is due to the increase of data transfers granularities (split and join actors) and to the decrease of posix execution system times. It can be observed that the downward trend of execution time is bounded. There is practically no difference between execution times of applications with partial and full reduction, despite that the full reduction generates a more complex application graph.

5. Conclusion

In this paper we have introduced the problem of parallelism reduction under throughput constraints. This problem arises in the domain of parallel computing and more specifically in dataflow programming.

We have proposed a generic parallelism reduction methodology. This methodology is based on equivalent actor partitioning, actor merge operations and on data stream time-multiplexing, time-demultiplexing. Although equivalent actor partitioning relies on a $\mathcal{NP}$-hard problem, we have shown that for real life applications it can be solved in polynomial time. Time-multiplexing and time-demultiplexing tools are used to respectively "join" and "split" streams of data to and from the merged actors. When compared to the parallelism reduction method based on pattern substitution proposed in [4] our method turns out to execute faster, predefined set of patterns is not needed and equivalent results are obtained by both methods. Nevertheless for some applications the pattern-based method provides some reductions of parallelism which our method is unable to do. In perspective we envisage to overcome this drawback.

The performance of the proposed methodology have been tested on a image processing algorithm - the Logarithm of Gaussian edge detection. The results of the tests point out that with enabled parallelism reduction the compilation times reduce, the generated binaries have smaller sizes and the execution performances are higher. The parallelism reduction can be applied on every single application: it preserves the application semantics and performance goals while reducing, if possible, the binary size. The worst case results in a non-modified application.

A limitation of this work is that only equivalent actors are merged, which is not always sufficient. In a future work we envisage to study the problem of parallelism reduction with non-equivalent actors merge.

References

[1] W. Najjar, E. Lee, G. Gao, Advances in the dataflow computational model, Parallel Computing 25 (1999) 1907–1929.

[2] G. Bilsen, M. Engels, R. Lauwereins, J. Peperstraete, Cyclo-static dataflow, Signal Processing, IEEE Transactions on 44 (2) (1996) 397–408.

[3] T. Goubier, R. Sirdey, S. Louise, V. David, ΣC: A Programming Model and Language for Embedded Manycores, in: Algorithms and Architectures for Parallel Processing, Vol. 7016 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2011, pp. 385–394.

[4] L. Cudennec, R. Sirdey, Parallelism reduction based on pattern substitution in dataflow oriented programming languages, in: Proceedings of the 12th International Conference on Computational Science, 2012.

[5] W. Thies, M. Karczmarek, S. Amarasinghe, Streamit: A language for streaming applications, in: Proceedings of the 11th International Conference on Compiler Construction, 2002, pp. 179–196.

[6] M. Gordon, W. Thies, S. Amarasinghe, Exploiting coarse-grained task, data, and pipeline parallelism in stream programs, in: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems, ASPLOS-XII, 2006, pp. 151–162.

[7] J. Cong, W. Jiang, Pattern-based behavior synthesis for fpga resource reduction, in: Proceedings of the 16th international ACM/SIGDA symposium on Field programmable gate arrays, FPGA '08, 2008, pp. 107–116.

[8] J. Cong, H. Huang, W. Jiang, A generalized control-flow-aware pattern recognition algorithm for behavioral synthesis, in: Proceedings of the Conference on Design, Automation and Test in Europe, DATE '10, 2010, pp. 1255–1260.

[9] S. O. Memik, G. Memik, R. Jafari, E. Kursun, Global resource sharing for synthesis of control data flow graphs on fpgas, in: Proceedings of the 40th annual Design Automation Conference, DAC '03, 2003, pp. 604–609.

[10] S. Carpov, J. Carlier, D. Nace, R. Sirdey, Task ordering and memory management problem for degree of parallelism estimation, in: Lecture Notes in Computer Science, Vol. 6842, 2011, pp. 592–603.
[11] E. G. Coffman, Jr., M. R. Garey, D. S. Johnson, Approximation algorithms for bin packing: a survey, PWS Publishing Co., 1997, pp. 46–93.
[12] M. R. Garey, D. S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, W. H. Freeman & Co., 1979.
[13] R. Gonzalez, R. Woods, Digital Image Processing, Addison-Wesley Longman Publishing Co., Inc., 2001.